# Assembly language Lab 1

We will be using SASM (<https://dman95.github.io/SASM/english.html>) to build and test our assembly language programs. This is an integrated IDE with a couple of assemblers and debugger built in.

SASM has builds for Windows and Linux, and there is a link to build it under Mac OS on the website.

The assembler should be defaulted to NASM, which Is the same assembler that the Duntmann book is using so almost all the assembly in the book can be straight translated to SASM, however being a cross-platform environment it uses special macros to perform simple I/O rather than using the straight Linux calls in Duntmann.

Note that in our examples and the Duntmann book the code is written in Intel style e.g.  
**MOV destination, source**  
equivalent to destination = source  
Math functions work the same way  
**ADD destination, source**  
equivalent to destination = destination + source

## Hello World!

Start SASM and load HelloWorld.asm

Run it by pressing the green arrow . This will assemble, link and run the program and in the output window you should see ‘Hello World!’

Let’s have a look at the code

%include "io.inc"

This is like #include in C, and it is importing the definitions for us to be able to read and write text

section .data

msg db 'Hello, world!', 0

The assembler is told that we will be using the data section of the program. This is an area that is predefined with data and can be read to/written from but you cannot execute code inside.

msg is defined as a label, and contains a sequence of bytes (db) and then we have the message, with a real integer 0 at the end (not a character 0) to signify the end of the string – a standard C-style string.

Whenever we use msg in our assembler from now on, it is substituted with the address of that label.

At the time of assembly/linking we do not know where any of this will be stored, only the offset into the .data, .text or .bss sections. The actual placement in physical memory will be left to the operating system and it fixes up all references when the program is loaded.

section .text

We switch to section text. This is read only and the only section that code can execute

global CMAIN

We tell the assembler that we will be creating a global label called CMAIN. This is required for the label to be exported out of the source file we are assembling. The linker is looking for CMAIN so it can be called to start our program.

CMAIN:

CMAIN is now defined as a label here. We need to put the colon after because there is no other information in the line and without the colon the assembler will give us a warning just to make sure that is what we wanted to do!

mov ebp, esp

Sets up a stack frame for the debugger. Don’t worry about this for the time being.

Moves the stack pointer (esp) into the base pointer (ebp)

PRINT\_STRING msg

NEWLINE

Here we are using macros from inside the io.inc file. PRINT\_STRING takes the address of a C style string (list of characters followed by a 0) and prints it. NEWLINE prints out a platform specific new line.

xor eax, eax

Equivalent of mov eax,0 - but why do it this way?

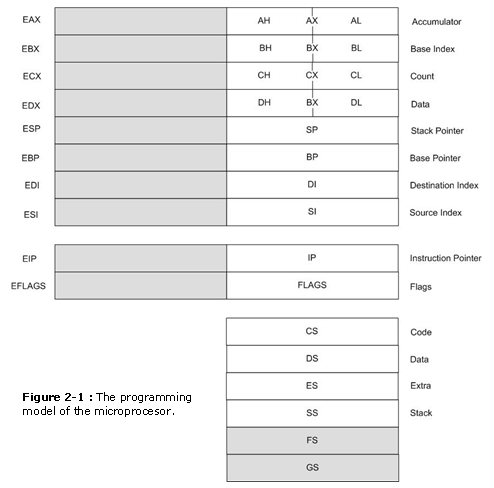
This loads eax with 0, which will be the return value from the function, and the program

ret

Return from the main function

## 80386 Registers

The basic integer registers of the 80386 are

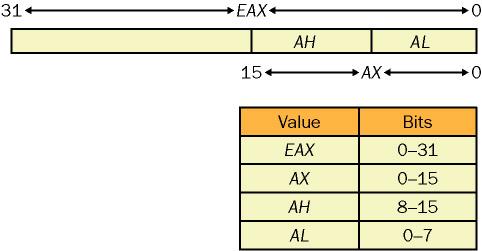


From these we only use EAX, EBX, ECX, EDX, ESI and EDI. These are all 32 bit wide.

ESP is the stack pointer, EBP is the ‘base’ pointer (used as a frame pointer in functions and for the debugger – we’ll cover that later)

EIP is the instruction pointer, EFlags is the flags register holding flags relating to the last arithmetic operation the processor did (such as Carry, Zero, Overflow etc)

The registers are split up into smaller sections than can access, 16 and 8 bit versions as below:

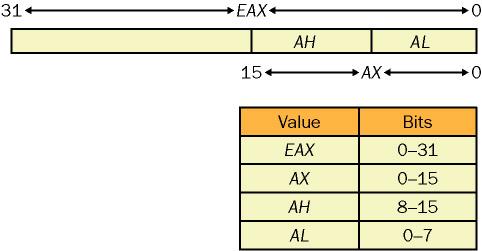


## Math1

Now load math1.asm and DO NOT RUN IT – yet. The code itself prints out a message, and loads some registers with values. It then performs some mathematical operations and writes out the result.

To start this task, I want you to predict what values it is going to print. Using a sheet of paper or perhaps an Excel spreadsheet mark out a grid with eax, ebx, ecx and edx (the registers) and space for their values.

Working your way down the code update the value box for each register and also make a note of what value would be written out with the PRINT\_DEC line

Note the PRINT\_DEC line has a 4 before the register we want to print out. This is specifying how many bytes to use to calculate the number. As we are using eax,ebx etc they are 32 bit values – i.e. 4 bytes.   
All of our registers are broken up in the following way, into 8 bit, 16 bit, 32 bit and 64 bit (RAX) if we were using 64 bit assembler  


The instructions we are using and the C equivalents are:

mov destination, source

destination = source

add destination, source

destination = destination + source

sub destination, source

destination = destination – source

inc something

something = something + 1

sub something

something = something – 1

imul destination, source

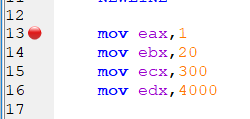
destination = destination \* source

Once you have written out all the results you think the program will print, run the program and check your answers. If you got it wrong, go back and double check your results.

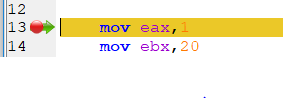
### Debugging

Now we will run the program in the debugger to see the changes as they happen.

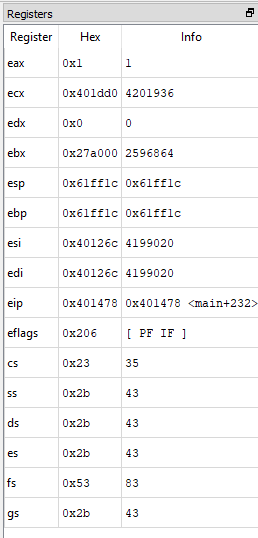
Click to the right of line 13 to add a breakpoint



Then press the debug button . The program will stop at the first line of CMAIN, so press this button again to continue and it will stop at line 13.



Now go to the Debug menu and select Show Registers to show this on the right hand side



(Actual values will vary at this point. And for some odd reason ebx is under edx!)

Now click the Step Over button  to step through the code. Notice that eip (the instruction pointer) increases as we move through the program.

As the registers are loaded and mathematical operations performed, observe the registers changing.

### Your turn!

Open up mymath1.asm. We are going to use that as a template to write some math code.

We are going to calculate

a = (b+c) \* 7 \* (d-6)

and print out the result for a

So far we have only used registers on both sides of our arithmetic instructions, but we can use an *immediate value* as the source, e.g.:

add ebx, 3

Keep b, c and d in registers at the start of the program and solve for the following sets of data by initialising the registers at the start of the code with new variables, and running it. Check your results.

b=16, c=3, d=1

b=-99, c=17, d=230

b=4, c=-13, d=6